TIPMerge: Recommending Experts for Integrating Changes across Branches

Catarina Costa[1,2], Jair Figueiredo[1]
[1]Federal University of Acre
Rio Branco - AC, Brazil
{catarina,jjcfigueiredo}@ufac.br

Leonardo Murta[2]
[2]Fluminense Federal University
Niteroi – RJ, Brazil
leomurta@ic.uff.br

Anita Sarma[3]
[3]Oregon State University
Corvallis, USA
anita.sarma@oregonstate.edu

ABSTRACT

Parallel development in branches is a common software practice. However, past work has found that integration of changes across branches is not easy, and often leads to failures. Thus far, there has been little work to recommend developers who have the right expertise to perform a branch integration. We propose TIPMerge, a novel tool that recommends developers who are best suited to perform merges, by taking into consideration developers' past experience in the project, their changes in the branches, and dependencies among modified files in the branches. We evaluated TIPMerge on 28 projects, which included up to 15,584 merges with at least two developers and potentially conflicting changes. Our results show that TIPMerge is successful. On average, 85% of its top-3 recommendations correctly included the developer who actually performed the merge. Best (accuracy) results of TIPMerge recommendations were at 98%. Our interviews with developers of two projects reveal that in cases where the TIPMerge recommendation did not match the actual merge developer, the recommended developer had the expertise to perform the merge, or was involved in a collaborative merge session.

Categories and Subject Descriptors

D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement – *version control*;

General Terms

Design, Experimentation, Human Factors, Management.

Keywords

Version Control, Branch Merge, Expertise Recommendation.

1. INTRODUCTION

Parallel development is a common practice to manage time to market, isolate new features from bug fixes, segregate development teams, implement customizations, etc. Branching is the most commonly adopted mechanism to support parallel development for code under version control [7, 32].

Changes made in branches need to be reintegrated periodically through a merge operation. This operation combines two independent, and usually long sequences of commits, which can potentially hold numerous contributions from different developers. For instance, in previous work [11, 12] we observed a merge in the Rails project (https://goo.gl/7fP3fv), which included commits made by 47 developers in one branch and 52 developers in the other. In fact, our data from 28 projects show that on an average 29.14% (median 29.67%) of such merges involved changes from at least three developers. And such merges occurred frequently, around every 2 days (median).

Integrating changes across branches is not easy. In a Stack Overflow discussion (http://goo.gl/uMvZHk), a developer laments: "*when trying to merge the changes on the trunk with a branch, there are conflicts on 10 different files, which are authored and maintained by 3 different developers*." Merging branches can be difficult because of several reasons. First, conflicts can arise, especially when long-living branches are merged [6]. In fact, Shihab et al. [32] found that the adoption of branches can cause integration failures due to conflicts or unseen dependencies. Second, when conflicts do occur, it is not always clear which changes to keep and which to reject. The developer performing a merge might not fully understand the changed code or the rationale behind the change, or may not have the expertise to determine the impact of the change since they do not fully understand the dependencies in the project [11].

Unfortunately, existing support for integrating branches is rudimentary. Most tools usually detect only direct (i.e., textual) conflicts, and transfer the responsibility of resolving conflicts to the developer in charge. In complex merge situations, developers might not always know how to make the right decision. For instance, a survey with 164 developers [11] showed that people frequently needed to make decisions with which were comfortable when performing a merge. This is likely a reason for developers performing collaborative merge sessions [20, 21, 26] .

However, identifying the appropriate developers to perform a merge is not an easy task either. On the one hand, inviting all involved developers to a merge session is infeasible due to cost, physical space, and developer availability. On the other hand, inviting just one (or two) developer to the merge session requires enough knowledge about the project to be able to prioritize among those developers who are aware of the project history, the dependencies in the project, and the changes made in the branches.

Recent work has investigated developer recommendations to analyze pull request [18, 22, 36, 37]. However, these approaches do not translate well to branch integration. While pull requests refer to remote lines of development that need to be merged, these "branches" usually contain few commits by a single developer [14]. Moreover, the author of the pull request usually syncs their forked branch in advance to ease reintegration, making the process more like a workspace commit. In our more general case of branch integration, the number of developers, the time interval between syncing of the branches, and the number of commits per branch varies, and can be very high, making the process a lot more complex.

In this paper, we propose TIPMerge, a novel tool that identifies the most appropriate developers to merge branches. For a given pair of branch, TIPMerge first identifies "key" files and the developers who have made changes to them in each branch. Key files are files that are changed in parallel across the branches (which can lead to direct conflicts), or files that have changed in one branch, but have dependencies with other changed files in the other branch (which can cause indirect conflicts). TIPMerge then identifies overall experience of developers with the key files based on the project and branch history. After analyzing this information, TIPMerge recommends a ranked list of developers who are best suited to integrate a pair of branch.

To empirically evaluate the usefulness of our approach, we measure the accuracy of the recommendations. We use *top-1* and *top-3* accuracy as the likelihood that the correct developer is in the first k (1 or 3) recommendations. We also measure the *normalized accuracy improvement* over the majority class – the developers who have done most of the merges. Our corpus was composed of 28 software projects, which included 15,584 merges with at least 2 unique developers and potentially conflicting changes (i.e., with key files). On average, 85% of the top-3 recommendations made by TIPMerge correctly included the developer who actually performed the merge. The best accuracy (98%) was obtained in the Diploma project. Moreover, in 82% of the merges, TIPMerge obtained higher accuracy than selecting the developer who performed most of the previous merges (i.e., the majority class).

To better understand the cases were TIPMerge made incorrect recommendations, we interviewed developers from two of the projects. In several of these cases, the developers agreed that the TIPMerge recommendation was also valid. In some cases, the developers ceded that TIPMerge recommendation was more appropriate. In other situations, we found that the recommended developer had, in fact, participated in a collaborative merge.

This paper makes the following contributions:

- **Approach**. We present a novel approach that analyzes change history in branches, file dependencies, and the past history to recommend expert developers to merge branches.
- **Implementation**. We implemented our approach in a tool that uses a medal-based ranking system to recommend developers. It is built in Java and analyzes projects in Git.
- **Empirical Evaluation**. We present results of our mixed method evaluation. We quantitatively evaluated 28 real-world projects to show that TIPMerge has high normalized accuracy improvements over the majority class in its recommendation: top-1 recommendation in Lantern (49.70%) and top-3 recommendations in Diploma (82.39%). Our interviews show that different factors (e.g., development role, skills, past collaboration) affect who actually performs the merge.

2. TIPMERGE

The primary goal of TIPMerge is to recommend developers with the expertise to merge changes across two branches. We do so by leveraging the project history. More specifically, our approach has the following steps:

1. Extract data from the repository until the branch tips. That is, the two most recent commits of the two branches that will be merged.
2. Detect dependencies among files by identifying files that were frequently co-committed (logical coupling). We calculate dependencies from the data before the branch creation.
3. Identify developers who edited key files – files that were edited in both branches or had dependencies across branches (see Section 2.4). We collect this information for changes in branches as well as previous history.
4. Recommend a ranked list of suitable candidates to perform the merge based on a medal count system (see Section 2.5).

2.1 Scenario

Before describing our approach, we present an intentionally simple scenario to illustrate the use of branches. Let us consider a hypothetical project Calculator, which employs a feature branch in parallel to the master branch to implement advanced operations. Figure 1 presents a commit history that includes these two branches, and four developers: Alice, Peter, Bob, and Tom. Let us assume that Bob creates a feature branch from the master (C50) and performs three commits (C51, C54, and C56). Tom also commits to this branch (C57). Alice and Peter continue to work in the master branch in parallel. Alice performs two commits (C52 and C53), followed by two commits from Peter (C55 and C58). Let us further assume that Alice and Bob change the same files, *QuadraticEquation* and *Subtraction*, across the branches (see Table 1 and Table 2). Peter changed files *Multiplication* and *Division* in the master branch. Tom changed only file *IEquation* in the feature branch. However, there is a logical dependency: file QuadraticEquation depends on file *IEquation*.

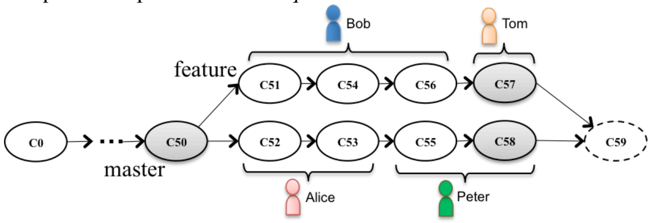


Figure 1. Example of Merging Branches

In our example, developers are unaware of changes made in another branch. Therefore, Alice does not know about the parallel changes made by Bob to *QuadraticEquation* and *Subtraction* in the feature branch. A merge of the branches will generate a merge error due to direct conflicts. Further, Tom changed *IEquation* in the feature branch, on which *QuadraticEquation* depends, and is changed by Alice in the master branch. A merge of these branches can generate build or test failure due to indirect conflicts.

Additionally, Table 3 shows (a hypothetical) edit history of the project files before the branching. Alex had edited all the five files and Anna four of the five files.

Table 1. Commits in the master branch

File Name	Alice	Peter
QuadraticEquation	2 (C52, C53)	0
Subtraction	1 (C53)	0
Multiplication	0	2 (C55, C58)
Division	0	2 (C55, C58)

Table 2. Commits in the feature branch

File Name	Bob	Tom
QuadraticEquation	2 (C51, C56)	0
Subtraction	3 (C51, C54, C56)	0
IEquation	0	1 (C57)

Table 3. Contributions in history before branching

File Name	Alice	Bob	Tom	Alex	Anna
QuadraticEquation	3	0	0	11	4
Subtraction	0	2	0	3	0
Multiplication	0	0	0	4	2
Division	0	0	0	1	3
IEquation	0	0	4	6	2

We analyze information about changes across branches as well as the previous history because both are relevant for merging branches. Developers who have made changes in the branches know about recent changes that need to be integrated. Developers who have modified files in the past may know about the history and goals of the implementation.

2.2 Data Extraction

The first step in our approach is extracting the data about branches from the projects. Formally, we can define a project p as a tuple (F, D, C), where F is a set of files, D is a set of developers, and C is a set of commits. Each commit $c_i \in C$ is a tuple (F_i, a_i, P_i), where $F_i \subseteq F$ is the set of files changed (add, remove, or edit) by c_i; $a_i \in D$ is the author of c_i; and $P_i \subset C$ is the set of parent commits of c_i (Figure 2).

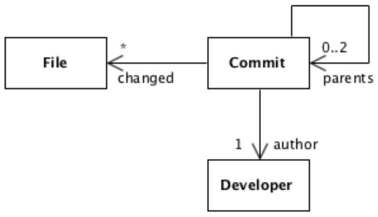


Figure 2: Simple versioning metamodel

Commits are organized in a directed acyclic graph (e.g., see Figure 1), where the first commit of the project has no parent (e.g., commit C0 in Figure 1), *revision* commits have only one parent (e.g., commit C53 in Figure 1), and *merge* commits have two parents (e.g., commit C59 in Figure 1). All reachable commits from c_i form its history, including c_i itself and the transitive closure over its parents. In Figure 1, {C0, …, C51, C54, C56, C57} is the history of commit C57. The *history* of $c_i \in C$ is defined as:

$$H_i = \{c \in C \mid c = c_i \vee \exists p_j : (p_j \in P_i \wedge c \in H_j)\}$$

Two commits $c_i, c_j \in C$ that do not reach each other (i.e., $c_i \notin H_j \wedge c_j \notin H_i$) are called *variants* (e.g., commits C57 and C58 in Figure 1). Variants may have a common history, which comprises all commits that exist in both histories. In Figure 1, {C0, …, C50} is the common history of commits C57 and C58. The *common history* of $c_i, c_j \in C$ is defined as:

$$CH_{i,j} = H_i \cap H_j$$

The history of each variant also comprises commits that do not belong to the common history, forming an independent line of development called branch history. For example, {C51, C54, C56, C57} is the branch history of C57 when merging with C58, and {C52, C53, C55, C58} is the branch history of C58 when merging with C57 (Figure 1). As branches can be created from other branches, the branch history may vary depending on the opposing branch, as a consequence of different common histories. The *branch history* of $c_i \in C$ when merging with $c_j \in C$ is defined as:

$$BH_{i,j} = H_i \backslash H_j$$

Each branch history comprises a set of files changed by its commits. The *files changed in the branch history* of $c_i \in C$ when merging with $c_j \in C$ is defined as:

$$F_{i,j} = \bigcup_{c_k \in BH_{i,j}} F_k$$

In addition, *file edited in the common history* (i.e., $\bigcup_{c_k \in CH_{i,j}} F_k$) is also extracted to determine expertise over the key files. Currently, we collect data of all past commits, but the approach can be easily modified to only consider changes in a given time frame (e.g., past release) to accommodate decay in expertise [33].

2.3 Dependency Detection

Next, we identify dependencies among files that are edited across branches. This is vital, since parallel changes to dependent files can cause indirect conflicts when the branches are integrated.

There are two different ways to identify dependencies ([27, 33, 38, 39]): using program analysis or logical coupling. Dependencies detected via program analysis typically identify structural or syntactic dependencies. However, such analysis techniques are language dependent. Logical coupling, on the other hand, detect evolutionary dependencies by identifying files (or code) that are frequently changed together [27], and is language agnostic. A majority of open source projects involve different languages and many times a combination of different programming languages. Therefore, we use logical dependencies in our approach.

We use the edit history of the project (before the branching occurred) to determine dependencies between pairs of files. Of course, it is possible that these dependencies might change based on edits in the branches themselves. However, the past history provides us a baseline of these dependencies. In future work, we will investigate how dependencies change from the baseline because of change in branches and their effect on recommendations.

Further, since our goal is to identify the expertise for branch merging, we only consider the impact of changes to dependent files across branches. We assume that all commits within the same branch have already been integrated. For example, in our scenario, we assume that Peter has already integrated the changes made by Alice prior to making his commits as they are working on the same (master) branch.

To understand how we compute the *logical dependencies across files*, lets assume that each file $f_l \in F$ has a set of dependencies $Dep_l \subset F$ that are obtained by using an *association rule* mining technique. An association rule is a pair (X, Y) of two disjoint entity sets $X, Y \subset F$. In the notation $X \rightarrow Y$, X is called antecedent and Y is called consequent [1]. It means that, when X occurs, Y also occurs, even if they are not structurally related [27]. However, its probabilistic interpretation is based on the amount of evidence in the transactions from which they are derived [39]. This amount of evidence is determined by two metrics: (1) support – the joint probability of having both antecedent and consequent, and (2) confidence – the conditional probability of having the consequent assuming that the antecedent is present [1].

The confidence value can range from 0 to 1, where 1 means that every time that the antecedent is changed, the consequent is also changed. In this case, the use of a threshold is necessary because low confidence implies low probability that changing a file causes impact in the dependent file. Therefore, the use of confidence (instead of support) allows us to define direction in the dependencies. Development teams have the freedom to decide the threshold above which a dependency becomes relevant. Our approach parameterizes the threshold and can use the value set by the user. Here, after some empirical tests, we have chosen a confidence threshold of 0.6 to determine dependency.

In our scenario, we have dependencies between the files *QuadraticEquation* and *IEquation*. *IEquation* was changed in 12 commits, and let us assume that of these 12 commits, 8 also included changes to *QuadraticEquation* (Table 3). The confidence of the association rule (*IEquation →QuadraticEquation*) is 8/12 = 0.66. Since we are using a threshold of 0.6, we say that *QuadraticEquation* depends on *IEquation*. As confidence is not symmetric, the confidence value of the rule *QuadraticEquation →IEquation* can be different. In our scenario, *QuadraticEquation* was changed in 18 commits, and of these 18 commits, 8 also included changes to *IEquation*. The confidence of this rule is 8/18 = 0.44. Therefore, *IEquation* does not depend on *QuadraticEquation*.

2.4 Key File Author Identification

The next step in our approach is to identify the developers who have modified files that are relevant to the merging of the branches. We term these files as key files. These are files that have been changed in parallel in both branches (e.g., *Subtraction* and *QuadraticEquation*) or files that were changed in one branch (e.g., *IEquation*), but have a dependency with files that were changed in the other branch (e.g., *QuadraticEquation*) – both the dependent and the file causing the dependency are considered as key files. Changes to the former class of files can cause a merge failure (direct conflicts), whereas changes to the latter class can potentially lead to test or build failures (indirect conflicts). Only key files are relevant for us, as all other files can be automatically merged safely. Files that were unchanged in either branch are irrelevant for the merge. **Key files** are defined as:

$$KF_{i,j} = \left\{ f_k \in F \,\middle|\, \begin{array}{c} (f_k \in F_{i,j} \cap F_{j,i}) \vee \\ (f_k \in F_{i,j} \wedge Dep_k \cap F_{j,i} \neq \emptyset) \vee \\ (f_k \in F_{j,i} \wedge Dep_k \cap F_{i,j} \neq \emptyset) \vee \\ (f_k \in Dep_l \cap F_{i,j} \wedge f_l \in F_{j,i}) \vee \\ (f_k \in Dep_l \cap F_{j,i} \wedge f_l \in F_{i,j}) \end{array} \right\}$$

Once we have identified the key files, we are able to identify developers who may have changed these files: (1) in a branch, which signals expertise in the change, or (2) in the previous history, which signals expertise in the file.

In our scenario, the key files are *QuadraticEquation*, *Subtraction*, and *IEquation*. We see that Alice changed *QuadraticEquation* twice and *Subtraction* once in the master branch. Bob changed the same files in the feature branch: two and three times, respectively. Moreover, Tom changed *IEquation* once in the feature branch (see Table 1 and Table 2). When considering previous history (see Table 3), Alice changed *QuadraticEquation*, Bob changed *Subtraction*, and Tom changed *IEquation*. Further, Alex changed all the key files and Anna changed two of them (*QuadraticEquation* and *IEquation*).

2.5 Developer Recommendation

After identifying key files, our approach applies an algorithm that counts the number and type of contribution – changed in a branch or in the previous history – to recommend a ranking of suitable candidates who can perform the merge. We use a medal system to determine the position of each developer in the recommendation list. This is analogous to how countries are ranked in the Olympic Games based on medal counts. The following rules define when developers receive gold, silver, and bronze medals.

A *gold medal* is awarded when a developer changes a key file in a branch. The rationale is that the developer who changed a key file is the most knowledgeable about the change and its implications. They probably are also well versed with the file in general, and therefore, likely to be able to perform additional edits during a merge if necessary. *Gold medals* are defined as:

$$G_{i,j}(d) = \left| \bigcup_{c_k \in BH_{i,j} \wedge a_k = d} F_k \cap KF_{i,j} \right| + \left| \bigcup_{c_k \in BH_{j,i} \wedge a_k = d} F_k \cap KF_{i,j} \right|$$

A *silver medal* is awarded when a developer has changed a key file in the past. Developers who created or edited files in the past likely possess knowledge about the goals and requirements of these files, which can be helpful when performing a merge. *Silver medals* are defined as:

$$S_{i,j}(d) = \left| \bigcup_{c_k \in CH_{i,j} \wedge a_k = d} F_k \cap KF_{i,j} \right|$$

A *bronze medal* is awarded when a developer changes a file that depends on another file. The logic is that developers who have changed a dependent file, may have learned about the API (or methods) of the file that they are using. Consequently, they may know the goals and expectations of such a file, which may help in determining the impact of a change. *Bronze medals* are defined as:

$$B_{i,j}(d) = \left| \bigcup_{c_k \in BH_{i,j} \wedge a_k = d} \bigcup_{f_l \in F_k} Dep_l \cap F_{j,i} \right| + \left| \bigcup_{c_k \in BH_{j,i} \wedge a_k = d} \bigcup_{f_l \in F_k} Dep_l \cap F_{i,j} \right|$$

We assign a medal for each file edited, irrespective of the number of commits made. In our scenario, Alice and Bob each get one gold medal for *Subtraction*, even though Alice committed the file once in the master branch, and Bob committed it three times in the feature branch. Similarly, Bob and Alex each get one silver medal for *Subtraction,* because of their past changes (before branching). In our approach, we assume that when a developer edits a file, that developer has knowledge about the entire file. While our approach can support a finer-grained expertise calculation at the method level, we leave it for future work.

Our algorithm prioritizes developers with gold medals, because: (1) they are the expert on the change that has been made, and (2) they have the most recent knowledge about the file to which a change has been made. In the case of a tie in the number of gold medals, we use the number of silver medals to break the tie. This is because, everything being equal, a developer who has more experience overall is likely to be more suitable in merging changes. Finally, when there is a tie in the number of silver medals, we consider bronze medals. The notion is that if two developers have equal number of changes that they have made and equal knowledge of the project history, a developer who has additional knowledge about another file can be more suitable for the merge. This *medal ranking* is formally defined as:

$$R_{i,j} = \left(d_r \in D \,\middle|\, \begin{array}{c} G_{i,j}(d_r) + S_{i,j}(d_r) + B_{i,j}(d_r) > 0 \wedge \\ \left(\begin{array}{c} G_{i,j}(d_r) > G_{i,j}(d_{r+1}) \vee \\ \left(\begin{array}{c} G_{i,j}(d_r) = G_{i,j}(d_{r+1}) \wedge \\ \left(\begin{array}{c} S_{i,j}(d_r) > S_{i,j}(d_{r+1}) \vee \\ \left(\begin{array}{c} S_{i,j}(d_r) = S_{i,j}(d_{r+1}) \wedge \\ B_{i,j}(d_r) > B_{i,j}(d_{r+1}) \end{array} \right) \end{array} \right) \end{array} \right) \end{array} \right) \end{array} \right)$$

Table 4 shows that *QuadraticEquation* was changed by Alice and Bob, in the master and feature branches, respectively – earning them gold medals. Alice, Alex, and Anna also changed it in the previous history, each receiving a silver medal. *Subtraction* was changed by Alice and Bob in the branches, earning them a gold medal each. Bob and Alex get silver medals for editing *Subtraction* file in the previous history. Finally, Table 4 shows that only Tom modified file *IEquation* in the feature branch (earning a gold medal), and Tom, Alex, and Anna changed this file in the previous history (earning silver medals). Alice receives a bronze medal for *IEquation*, because she edited *QuadraticEquation.* Remember, file *IEquation* is a key file because *QuadraticEquation* depends on it and our assumption is that to be able to understand and edit the dependent file (*QuadraticEquation*), the developer must have some knowledge about the API (of in this case the interface *IEquation*).

Table 4. Medals (Gold | Silver | Bronze)

File	Alice	Bob	Tom	Alex	Anna
QuadraticEquation	1 | 1 | 0	1 | 0 | 0	0 | 0 | 0	0 | 1 | 0	0 | 1 | 0
Subtraction	1 | 0 | 0	1 | 1 | 0	0 | 0 | 0	0 | 1 | 0	0 | 0 | 0
IEquation	0 | 0 | 1	0 | 0 | 0	1 | 1 | 0	0 | 1 | 0	0 | 1 | 0

By counting the medals and tie-breaking when necessary, we generate a ranking of suitable candidates to perform the merge between a pair of branches. In our scenario (Table 5), Alice has the same number of gold and silver medals as Bob, but she has a bronze medal, which places her in the first position. Here, the first three candidates (Alice, Bob, Tom) all have gold medals. This implies that they each know about equal "amounts" of recent changes performed in the branches, and the tie breakers involving dependency information or past changes differentiate them.

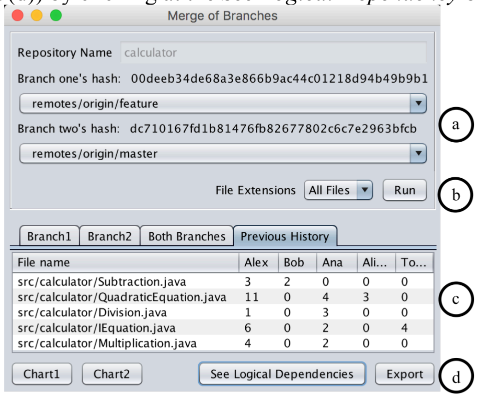
Table 5. Ranking of Candidates

Developer	Gold Medal	Silver Medal	Bronze Medal
1st Alice	2	1	1
2nd Bob	2	1	0
3rd Tom	1	1	0
4th Alex	0	3	0
5th Anna	0	2	0

3. IMPLEMENTATION

TIPMerge[1] is implemented in Java and is able to analyze projects versioned on Git, independently of their programming language[2]. We adapted Dominoes [33, 34] to identify logical dependencies among files across branches. Dominoes organizes data extracted from software repositories into matrices to denote relationships among software entities. For example, $[commit|file]$ denotes the files that were changed by commits in the project. These matrices are then combined to depict higher-order relationships, such as logical dependencies among files: $[file|file] = [commit|file]^T \times [commit|file]$.

To get the recommendation of developers to merge a pair of branches, the user first selects two branches to merge (Figure 3(a)) and triggers our recommendation analysis by clicking on the *Run* button (Figure 3(b)). Users can also filter file extensions that they know will be irrelevant to the recommendation. An obvious example can be *Readme.txt*. Once TIPMerge analyzes the project information, it shows for each developer the files that they have edited and the edit frequency in terms of commits (Figure 3(c)). This information is provided for each branch, both branches, and previous history. The user can also check the logical dependencies (Figure 3(d)) by clicking at the *See Logical Dependency* button.



Repository Name calculator

Branch one's hash: 00deeb34de68a3e866b9ac44c01218d94b49b9b1

remotes/origin/feature

Branch two's hash: dc710167fd1b81476fb82677802c6c7e2963bfcb

remotes/origin/master

File Extensions All Files

File name	Alex	Bob	Ana	Ali...	To...
src/calculator/Subtraction.java	3	2	0	0	0
src/calculator/QuadraticEquation.java	11	0	4	3	0
src/calculator/Division.java	1	0	3	0	0
src/calculator/IEquation.java	6	0	2	0	4
src/calculator/Multiplication.java	4	0	2	0	0

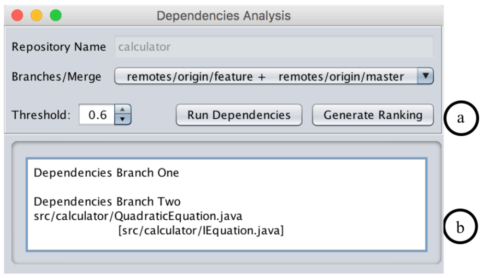
Figure 3. Information about changes and dependencies

In the *Dependencies Analysis* window (Figure 4), the user can configure the confidence threshold to visualize the logical dependencies among files (see Figure 4(b)). By clicking in the Generate Ranking button (Figure 4(a)), the user accesses the developers recommendation ranking.

Finally, using our medal-based algorithm, TIPMerge generates a ranked list of developers (Figure 5). For each developer (and each file), it lists the number of gold, silver, and bronze medals. It also shows the branch in which the change was made. Further information can be obtained through a tool tip, by hovering over the medal count. Figure 5(a) shows that Alice received a bronze medal for file *IEquation* because she changed *QuadraticEquation* in the opposite branch.

[1] https://github.com/gems-uff/tipmerge

[2]TIPMerge is language agnostic when analyzing expertise at the file-level. At the method-level, currently it only analyzes Java projects.



Repository Name calculator

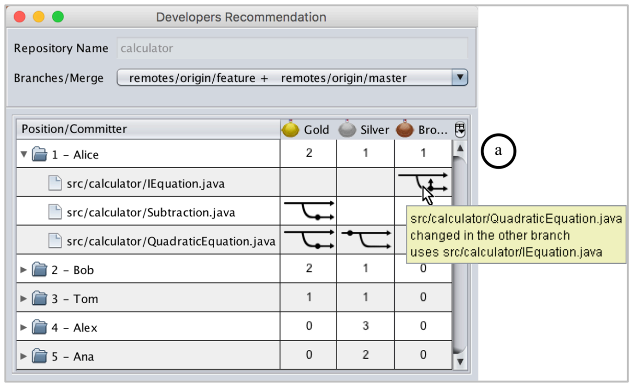
Branches/Merge remotes/origin/feature + remotes/origin/master

Threshold: 0.6

```
Dependencies Branch One

Dependencies Branch Two
src/calculator/QuadraticEquation.java
              [src/calculator/IEquation.java]
```

Figure 4. Files Dependencies



Repository Name calculator

Branches/Merge remotes/origin/feature + remotes/origin/master

Position/Committer	Gold	Silver	Bro...
1 – Alice	2	1	1
src/calculator/IEquation.java			
src/calculator/Subtraction.java			
src/calculator/QuadraticEquation.java			
2 – Bob	2	1	0
3 – Tom	1	1	0
4 – Alex	0	3	0
5 – Ana	0	2	0

src/calculator/QuadraticEquation.java changed in the other branch uses src/calculator/IEquation.java

Figure 5. Recommendation ranks for the project Calculator

4. QUANTITATIVE EVALUATION

To evaluate the recommendation provided by TIPMerge, we calculate the accuracy of its top-k recommendation, where k = 1 and 3. We select accuracy as the measurement metric, since our oracle includes just one element – the developer who actually performed the merge (henceforth, we call them *merge developer*). As a result, precision and recall is going to be 100% when we are correct, or 0% otherwise, and is not the best metric to use.

To evaluate the usefulness of our approach, we compare the accuracy of TIPMerge's top-k recommendation with the accuracy of choosing the top-k developers who performed the most merges in the past – the majority class (as commonly referred to in Machine Learning). The intuition is that we evaluate by how much our approach outperforms or underperforms as compared to a heuristic that picks the merge developer based on the total amounts of merges that a developer has previously performed.

4.1 Materials and Methods

We randomly selected over 100 projects from GitHub for analysis. For each project, we check: (1) whether the project includes merges, and (2) whether it comprises a sole developer performing a majority of the merges (>50%). The first criterion is self-explanatory. The second criterion is used to filter out those projects that either employ an integration manager or a small subset of developers who are responsible for performing the merge. For instance, the Git project has one developer who performed 9,385 out of 9,699 merges (96.76%). Such projects do not need a recommendation system, and are filtered out from the dataset.

After applying these criteria, we were left with 27 projects (see Table 6). In addition to these projects, we included another project – Diploma. Although this project has a developer who has performed 64% of the merges, we keep this project as we had access to the development team, which was useful for the qualitative analysis. Therefore, our final dataset comprised of **28 projects**. The median percentage of merges performed by the majority class in these projects was 29%.

Table 6. Selected Projects

Project	Language	Developers	Branches	Majority Class
Active Merchant	Ruby	402	26	20.34%
Akka	Scala	201	88	20.14%
Amarok	Ruby	196	2	20.71%
Angular	TypeScript	155	58	13.33%
Astropy	Python	142	11	25.94%
Cassandra	Java	103	8	24.04%
Comm-central	JavaScript	300	27	28.83%
Diploma	*Java*	*5*	*13*	*64.00%*
Errbit	Ruby	202	5	19.36%
Eureka	Java	36	5	40.00%
Falcor	JavaScript	21	16	44.74%
Firefox for iOS	C	40	286	21.69%
jQuery	JavaScript	227	4	45.20%
Katello	Ruby	61	16	13.16%
Khmer	Python	56	93	33.58%
Lantern	Go	48	67	22.83%
Maven	Java	45	23	47.06%
MCT	Java	13	5	44.80%
Nomad	Go	18	2	34.82%
Perl5	Perl	373	285	29.30%
Phoenix	Java	30	14	46.32%
PIConGPU	C++	12	3	39.52%
Priam	Java	27	14	44.04%
Sapos	Ruby	10	4	31.65%
Spree	Ruby	638	15	29.51%
Sympy	Python	385	4	28.76%
TYPO3	PHP	304	19	21.90%
Voldemort	Java	55	166	25.10%

Table 7. Selected Merges

Project	All Merges	Selected Merges	Percentage
Active Merchant	413	132	31.96%
Akka	5481	2189	39.94%
Amarok	396	198	50.00%
Angular	30	17	56.67%
Astropy	2386	855	35.83%
Cassandra	5762	4766	82.71%
Comm-central	111	30	27.03%
Diploma	250	156	62.40%
Errbit	532	125	23.50%
Eureka	620	108	17.42%
Falcor	342	100	29.24%
Firefox for iOS	779	205	26.32%
jQuery	250	132	52.80%
Katello	6890	2755	39.99%
Khmer	1087	473	43.51%
Lantern	1038	213	20.52%
Maven	34	13	38.24%
MCT	221	68	30.77%
Nomad	112	32	28.57%
Perl5	1826	733	40.14%
Phoenix	95	62	65.26%
PIConGPU	749	221	29.51%
Priam	302	97	32.12%
Sapos	139	85	61.15%
Spree	688	303	44.04%
Sympy	3647	1235	33.86%
TYPO3	210	50	23.81%
Voldemort	526	231	43.92%

Next, we identify the merges that would require a merge developer recommendation. That is, the merge is not simple: (1) it includes two or more developers, and (2) it includes changes to key files. Merges with key files can lead to direct or indirect conflicts, and therefore, may require higher expertise from the merge developer. For example, in Voldemort project, 231 of 526 merges (43.92%) included key files, and of these merges 64 faced direct conflicts. Based on these two criteria, we select **15,584 merges** from a set of 34,916 total merges (about 45%).

Next, we identify the merge developer for each of the (15,584) merges in our dataset. We then evaluate the prediction of TIPMerge to see whether the merge developer featured in the recommendation ranking. We specifically check 1st, 2nd, and 3rd position matches; we also keep tabs of higher order rankings (e.g., top-10 recommendation), or if the prediction completely missed the merge developer.

We then calculate the accuracy of TIPMerge recommendations for top-1 and top-3 recommendations. We recommend more than one developer since the most appropriate developer may not always be available (vacation, extensive backlog, etc.) or the merge developers may want to perform a collaborative merge session. We restrict ourselves to top-3 positions since we do not want to overwhelm the user with too many recommendations. Note, this makes our results conservative.

We then compare the TIPMerge top-k recommendations with the majority class based heuristic. That is, we compare the accuracy of top-1 recommendation of TIPMerge with the accuracy of using the top-1 majority class (the developer who performed the most merges). Similarly, we compare accuracies of TIPMerge top-3 recommendations with the top-3 in the majority class (the 3 most prolific merge developers). We use majority class as baseline because we are not aware of other approaches for recommending developers for merging branches. Moreover, without any additional information, a natural choice is to select someone who did a similar task (merges in our case) in the past.

However, directly comparing accuracies by their difference or direct proportion may lead to inflated results (>100% improvement), therefore, we use a measure for *normalized improvement in accuracy*. Figure 6 shows two scenarios where the accuracy difference between TIP (TIPMerge) and MC (majority class) is 10%. In the first scenario (Figure 6(a)), TIP is 100% more accurate than MC (20% vs. 10%). In the second scenario (Figure 6(b)), TIP is just 12% more accurate than MC (90% vs. 80%). If we simply calculate the difference in accuracies, it would indicate that both scenarios are equivalent. On the other hand, if we perform proportional comparison of accuracies, it would indicate a much higher increase in the first scenario (100% vs. 12%). Intuitively, it is clear that creating an algorithm that improves an already high majority class result by 10%, is much more difficult than improving on a low majority class result by the same amount. For instance, the room for improving over MC in the first scenario is 90% (from 10% to 100%) and TIP only reached 11% (10% ÷ 90%) of this potential. On the other hand, the room for improving over MC in the second scenario is only 20% (from 80% to 100%), but TIP achieved 50% (10% ÷ 20%) of this gain.

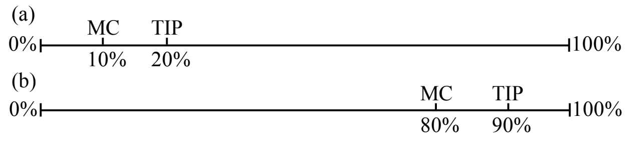


Figure 6. Examples of improvement in accuracy

We thus normalize the percentage of improvement in accuracy by considering "the room for improvement" by using f_p [28]:

$$f_p = \begin{cases} \dfrac{TIP_p - MC_p}{1 - MC_p}, if\ TIP_p > MC_p \\ \dfrac{TIP_p - MC_p}{MC_p}, otherwise \end{cases} \qquad \text{(Eq. 1)}$$

Where TIP_p represents the accuracy obtained by TIPMerge (top-1 or top-3) over project p, and MC_p represents the accuracy of the majority class (top-1 or top-3) of project p.

4.2 Results and Discussion

TIPMerge has been designed for situations where there is no integration manager or integration team, and the team would require recommendation about who should merge ranches. Therefore, we classify the results of our study into three categories:

Category I *(No integrators)***.** *Projects with majority classes (top-3) < 50%.* this shows that different developers perform the merge tasks. This is the context our approach was mainly designed for, as any developer is a potential candidate to merge branches.

Category II *(Integration team). Projects with majority classes (top-3) ≥ 50% and majority class (top-1) < 50%.* These projects don't have a single integrator (top-1<50%), but they have a group who perform the majority of merges. While not the primary audience of our approach, these teams might benefit since we can prioritize the most appropriate developers for the merge team.

Category III (*Integration manager). Projects with majority class (top-1) ≥ 50%.* These are project where there is a primary developer performing a majority of the branches. Our approach can be helpful to enable the lead integrator find developers for collaborative merge or help in integration.

To be conservative in our approach, we recalculate the majority class for the 15,584 merges in our dataset, and the percentage of merges performed by the majority class. Table 8 lists the accuracy of the top-1 recommendation by TIPMerge and the accuracy of the top-1 majority class. We also list the *normalized accuracy improvement* **(Eq. 1)** by TIPMerge. Table 9 provides similar data, but for top-3 rankings. These tables also color-code the improvement in accuracy for easier comprehension.

Category I: We note that TIPMerge has very good accuracy for projects in Category I for both top-1 and top-3 recommendations, except the Angular project. The project with the best improvement is *Lantern*. In this project, TIPMerge improves accuracy by 49.7%, and 76.6% over selecting the Majority Class developers. Note that the top-3 majority class performs 47.98% of the merges, which leaves about 52% of developers who perform merges otherwise. Even in such a situation, TIPMerge outperforms predictions using the majority class. Accuracy improvements (median) for the top-1 and top-3 recommendations (excluding the Angular project, which is discussed next) are at 28% and 47.7% respectively. This attests to the usefulness of TIPMerge in projects where there is diversity among merge developers.

In the Angular project, TIPMerge did worse than the prediction using the majority class (top-1 at -66.7%, top-3 at -12.5%). Indeed, TIPMerge correctly recommended only 1 and 7 merge cases (out of 17 total merges) for the top-1 and top-3 recommendations, respectively. On further investigation we find that in 9 of the incorrect recommendations, TIPMerge recommended the merge developer in other positions (i.e., we get an accuracy of 94% if we consider top-9 positions). To better understand the project dynamics, we investigate the merge developers forming the majority class. The top merge developer (alexeagle) had Continuous Integration (CI) experience; the second most prolific merge developer (alexwolfe) was the head of UX, and the third (yjbanov) was a Google employee who had been part of the project since the beginning. Therefore, in this case it is likely that alexeagle did most of the merges because of his CI background; alexwolfe and yjbanov, probably because of their knowledge of the project history and being part of the core team.

Table 8. Accuracies for the top-1 recommendation

Project	Majority Class	TIPMerge	Normalized Improv. Accuracy
Category I			
Lantern	20.66%	60.09%	49.70%
Katello	16.81%	50.60%	40.62%
Voldemort	23.38%	49.35%	33.89%
TYPO3	18.00%	36.00%	21.95%
Symply	21.70%	35.63%	17.79%
Active Merchant	21.21%	31.82%	13.47%
Angular	17.65%	5.88%	-66.69%
Category II			
Cassandra	24.63%	61.54%	48.97%
Eureka	36.11%	62.04%	40.59%
Akka	21.70%	48.79%	34.60%
Falcor	39.00%	60.00%	34.43%
Perl5	31.38%	54.71%	34.00%
Sapos	31.76%	54.12%	32.77%
Phoenix	40.32%	59.68%	32.44%
MCT	42.65%	60.29%	30.76%
Khmer	35.31%	55.18%	30.72%
Nomad	37.50%	53.13%	25.01%
Priam	30.93%	49.48%	26.86%
Errbit	21.60%	40.80%	24.49%
Spree	33.33%	48.51%	22.77%
Amarok	23.23%	39.90%	21.71%
Astropy	25.50%	35.79%	13.81%
Comm-central	46.67%	50.00%	6.24%
Firefox for iOS	39.51%	39.02%	-1.24%
jQuery	49.24%	30.30%	-38.46%
Category III			
Diploma	52.56%	58.33%	12.16%
Maven	84.62%	30.77%	-63.64%
PIConGPU	50.23%	16.74%	-66.67%

Table 9. Accuracies for the top-3 recommendations

Project	Majority Classes	TIPMerge	Normalized Improv. Accuracy
Category I			
Lantern	47.89%	87.79%	76.57%
Katello	41.52%	86.28%	76.54%
Voldemort	49.35%	83.55%	67.52%
TYPO3	44.00%	58.00%	25.00%
Symply	49.23%	62.11%	25.37%
Active Merchant	45.45%	60.61%	27.79%
Angular	47.06%	41.18%	-12.49%
Category II			
Cassandra	52.20%	79.98%	58.12%
Eureka	72.22%	94.44%	79.99%
Akka	55.55%	87.94%	72.87%
Falcor	90.00%	93.00%	30.00%
Perl5	69.30%	87.45%	59.12%
Sapos	68.24%	91.76%	74.06%
Phoenix	87.10%	87.10%	0.00%
MCT	73.53%	94.12%	77.79%
Khmer	64.90%	92.81%	79.52%
Nomad	75.00%	90.63%	62.52%
Priam	70.10%	91.75%	72.41%
Errbit	60.00%	70.40%	26.00%
Spree	61.39%	80.53%	49.57%
Amarok	51.01%	69.70%	38.15%
Astropy	63.27%	65.85%	7.02%
Comm-central	70.00%	90.00%	66.67%
Firefox for iOS	70.73%	85.85%	51.66%
jQuery	74.24%	69.70%	-6.12%
Category III			
Diploma	89.10%	98.08%	82.39%
Maven	100.00%	76.92%	-23.08%
PIConGPU	89.59%	75.57%	-15.65%

Category II: TIPMerge obtains high accuracy. In 16 of the 18 projects in this category, we obtain a higher accuracy than the majority class for top-1 recommendation, where the median improvement is at 30.7%. When considering top-3 recommendations, we have an improvement in 17 out of the 18 projects; median improvement is at 59.1%. We perform the best in the Cassandra project, with accuracy improvements at top-1, and top-3 recommendations at 49% and 58.1%, respectively.

Next, we investigate the two cases where TIPMerge had low accuracy: Firefox for iOS and jQuery. In the former case we get a low accuracy (39.02%) for the top-1 recommendation. However, we only have a decay of -1.2% from the majority class since we get the correct merge developer in 80 out of 205 merge cases, whereas the majority class performed 81 of the total merges in the project. When considering the top-3 recommendations, we have an accuracy of 85.9% (and an improvement of 51.6%). We investigate further into the project to determine why we missed one of the top-1 recommendation. We see that the top-3 merge developers (majority classes) in the project are st3fan, wesj, thebnich, and they are all Mozilla employees. Further, st3fan is the most senior core developer in the team. Therefore, it is likely that he possessed past project knowledge and had an idea about the project's future directions. This might be the reason for his performing most of the merges, which might not be reflected in our expertise calculation that weighs recent (branch) changes higher.

In jQuery, at the top-1 recommendation, we get an accuracy of 30.3% (leading to a -38.5% decay). Similar to "Firefox for iOS", we perform much better in the top-3 recommendations results (69.7% accuracy leading to a -6.1% decay). To better understand why the majority class prediction fared better, we investigate the team's contribution structure. The top-1 merge developer is jeresig, who was the founder and until recently had been the major contributor of the project. Therefore, it is likely that he was responsible for a large portion of the merges. The other two developers in the majority class are: (1) dmethvin, who is the president, a member of the board of directors, and a long-term contributor to the project, and (2) jaubourg, who is part of the core/standards team. Therefore, it is likely that dmethvin knew about the direction and goals of the project and was responsible for many of the merges; whereas jaubourg was responsible for merges because of his role in the standards (quality) team.

Category III: We were not expecting good results from projects in Category III, since these projects have a clear integrator. TIPMerge accuracies (top-3) for Maven and PIConGPU were at 76.92% and 75.7%, respectively. While such accuracy results are good by themselves, they were not an improvement over the majority class predictions, since they are very high themselves. Maven clearly has three developers responsible for merges with key files (responsible for 100% of the selected merges). PIConGPU has one developer responsible for most of the merges (50.23%), and three developers responsible for almost all merges (89.59%). These results confirm our assumptions that TIPMerge is not as useful for projects with clear integrators.

We were, however, surprised to have improvement in accuracy over the majority class in the project Diploma (12.2% and 82.4% for top-1 and top-3 recommendations). This project had a small development team (five), and the developers could physically meet with each, which might have led to the positive results.

In summary, our results indicate that TIPMerge provides very promising results for projects in Category I (no integrators) and Category II (integration team). When considering the top-3 recommendations, our approach has normalized improvements (median of 59.12%) in accuracy over the majority classes in 24 out of the 28 projects.

5. QUALITATIVE EVALUATION

To understand better why TIPMerge recommendations diverged, we performed a qualitative evaluation with two projects: one open-source (Sapos) and one proprietary (Diploma). We identified a set of previous merges where TIPMerge recommended a different developer than the person who performed the merge. We interviewed a few team members from each project to understand whether our recommendation was incorrect or whether there were other circumstances that affected the "merge developer" choice. In this section we discuss the design and results of this evaluation.

5.1 Materials and Methods

We selected Sapos and Diploma as our projects, since we had access to at least one team member who was extensively involved in branch merges. Diploma and Sapos (https://goo.gl/YKBnPw) had a team of 5 and 10 developers, respectively.

When considering merges with key files, Diploma and Sapos contained 156 and 85 merges, respectively. From this set, we selected for further analyses a set of merges which were complex and would cause a direct conflict. We selected a set of merges where: (1) TIPMerge provided an incorrect recommendation (the merge developer was not in the top-1 position), and (2) TIPMerge recommendation was in the correct position. This gave us 5 and 6 merge cases from Diploma and Sapos, respectively.

For each of these merges, we asked the interviewee to primarily: (1) reflect whether the merge developer was the most appropriate person in the project for the merge, and (2) evaluate the TIPMerge recommendation – top-1, as well as the top-3.

5.2 Results and Discussion

We interviewed one expert from Diploma and two experts from Sapos. These experts were the developers who performed the most merges (the majority class) in each project.

Diploma is a proprietary project developed by a government company in Brazil. It started in 2014 and comprises 5 developers: the project manager, who is also the technical lead and developer (D1); the business analyst, who is also a developer (D2), and three other developers (D3, D4, and D5). All team members work in the same building, but have different (physical) offices.

We interviewed the developer who did the most merges (D1), and asked him why their project uses branches and why he performed more than 50% of the merges. He revealed that the project used branches to maintain system integrity. Four branches were specified: development, staging (acceptance tests), production, and hotfix. Additional branches were used to implement new requirements or test new technology. Regarding the number of merges that he performed, D1 said: "*I am the technical lead, I have more working hours, and I take care of approval and production. I have to maintain the integrity of this structure. I have to help the team and I produce more*". He added that, in case of conflicts where there is no clear merge decision, he contacts the developer who made the change and performs a collaborative merge. Besides that, when another developer has difficulties in merging, D1 is always available to pair and provide support.

We presented to the developer: (a) two cases where TIPMerge recommended the merge developer in the 3rd position, (b) two cases where we recommended the merge developer in the 2nd position, and (c) one case where we recommended the merge developer in the 1st position (see Table 10).

In the first merge case, D4 performed the merge (in bold in Table 10), but our approach placed D1 in the 1st and D4 in the 3rd position. We asked D1 whether he could have performed the merge. He said: "*It makes sense... I help him in the merge... D4 must have been the one to do the merge ultimately because he was* *the last to commit."* In the second case, D1 performed the merge, whereas our approach suggested D4 in the 1st, and D1 in the 3rd position. We asked D1 about this, and he said: *"D4 had changed two tasks, but there is a piece of code in the merge that only I know, so [I did the merge]… but D4 would also have been able to perform it."*

Table 10. TIPMerge ranking and the developer who merged (in bold)

TIPMerge Position	Diploma	Sapos
3rd	D1, D5, **D4**	D4, D5, **D3**
	D4, D2, **D1**	D4, D5, **D3**
2nd	D3, **D2**	D1, **D2**
	D1, **D3**	D1, **D2**
	-	D4, **D1**
	-	D4, **D1**
1st	**D1**	-

Next, we investigate instances where TIPMerge recommended the developer who performed the merge in the 2nd place. In one of the cases, D2 performed the merge and our approach suggested him in the 2nd place. Our interviewee (D1) said: *"They (D2 and D3) were… in parallel. I think they had the same knowledge. Maybe I would have chosen D2 because he had made some of these changes with me. But I think any of them would have been able to perform this merge"*. In the fourth case, D3 performed the merge, whom we ranked in the 2nd position. When asked about this, D1 said: *"I would still have helped him in this merge. While D3 could have perform [the merge], I would have followed it closely."* Note, we ranked D1 in the 1st place.

In the last merge instance, we selected a merge with a conflict that D1 performed, and for which our approach recommended him in the 1st position. We asked him to check whether he was in fact the only one who could have performed this merge and he answered: *"Yeah, as there were some parts of a legacy system and only I know this part, I should indeed have done this merge*".

In summary, we see that in cases where TIPMerge recommendations were not in the top positions, the merge decision could have been based on: (1) the person who had made the last commit and not necessarily with the most expertise, (2) special knowledge about a certain piece of code or parts of a legacy system, and (3) personal preference because of having collaborated with someone in the past. In some cases, while the top-1 recommendation by TIPMerge was not officially the merge developer, they were, in fact, involved in a collaborative merge. In none of the cases, did the interviewee say that the top-1 recommended developer would have been unable to perform the merge. Finally, the interviewee suggested that he would consider using TIPMerge in his project.

Sapos is an open-source project targeted at the management of information related to graduate programs. Ten developers (D1, …, D10) worked on the project at different time periods. We interviewed the two developers who did most merges: D1 and D3.

In Sapos, we selected 34 merge cases that had direct conflicts. In 9 of these merges, our top-1 recommendation was not the merge developer. We randomly selected 6 out of these 9 cases for further analysis (see Table 10). In the first two cases, the merge developer was ranked in the 3rd position, and in the remaining cases in the 2nd position.

D3 performed the merges in the first two cases; we ranked him in the 3rd position. We interviewed D3 and asked him, whether D4 (top-1 recommendation) would have been appropriate in both merges. He replied that D4 was actually the main author of these merges and they had worked collaboratively, but using D3's computer: *"We did these merges together in my office"*.

We interviewed D1 about the next four cases. D2 performed the first of these two merges, whereas we ranked him in the 2nd position. According to D1, both of them (D1 and D2) had worked together (pair-programed) extensively in the past, and thus they had equivalent knowledge of the project. Therefore, both were qualified to perform these merges. D1 performed the other two merges, and we ranked him in the 2nd position. According to D1, in both cases a merge conflict occurred in the database schema file. He was responsible for the merge because he added a database migration file to the branch. However, he said that D4 would be able to do the merge by analyzing the database migration file: "*He would need only to see the added and removed fields in each branch*".

In summary, in 66% of the cases Sapos developers have worked together in pairs (33% during the merge and 33% in the past). It seems that collaborative practices like pair programming can effectively propagate knowledge among developers, providing direct benefits for knowledge-intensive tasks like merge.

6. THREATS TO VALIDITY

As in any study, our study has limitations. First, in our evaluations we used the developer who had performed the past merge as our oracle (the most appropriate developer). This has been a common approach in work on expert identification [19, 22, 25]. However, it is possible that that developer was not the most appropriate developer. We ameliorated this issue by interviewing three developers from two projects to determine the appropriateness of our recommendation. Second, our approach uses the committers' git ID to identify developers. It is possible that developers use multiple aliases. We manually verified the TIPMerge ranking with the merge developer to fix possible mistakes by considering their ID similarity. Although this suffices in most cases, we may have missed some when the aliases are lexically different. However, note that if we did miss some aliases, they would in fact decrease the accuracy of TIPMerge results.

In our study, we checked for merges with at least two unique developers to avoid cases where a single developer was making parallel changes. However, our dataset still contains merges with only two unique developers. In these cases, the merge of the two branches is akin to a workspace merge, which is a simpler scenario than branches with a large number of contributors. In the future, we plan to perform a sensitivity analysis to determine the effects of the number of developers in a branch and in a merge on the TIPMerge recommendations.

In terms of generalizability of our results, we had five projects in Category I and we spoke to experts from two projects. In the future, we plan to replicate our results on a larger corpus and speak to more developers across different projects.

7. RELATED WORK

To the best of our knowledge, there is no work that addresses recommendation of developers to merge branches. The more closely related works either provide awareness to developers during parallel work to reduce the complexity of merges, or support the identification of experts in software projects.

7.1 Workspace Awareness

Research on workspace awareness aims to notify developers about parallel ongoing work and emerging potential conflicts that developers will face when they synchronize their work with the main development. Approaches such as CASI [29], CloudStudio [13], CoDesign [4], Crystal [8], Palantír [30], SafeCommit [35], Syde [16], and WeCode [15] try to avoid conflicts by notifying the developers and prodding them to self-coordinate. One of the most recognized approach on awareness is Palantír. It tracks workspace edits to identify potential conflicts and notifies developers of these conflicts as soon as possible. Similarly, Crystal integrates ongoing parallel changes, extracted from local commits (in git), into a shadow master repository to identify potential conflicts. CloudStudio allows a developer to select the type of information about parallel changes that they want to be notified about. This helps with interruption management. SafeCommit identifies changes at different levels of safety (will pass tests, will pass merge, etc.), thereby allowing developers the flexibility to choose which change sets can be safely integrated with the trunk.

Even though these approaches play an important role in minimizing the incidence of conflicts, they do not guarantee conflict-free merges. Different factors may still lead to difficult merges even when these approaches are in place, such as: developers working on project forks that eventually need to be reintegrated; (2) the nature of some parallel changes (e.g., bug-fix and new features over the same component); and (3) offline changes. In these cases, the integration process would impose challenges to the developers in charge and our approach would be useful.

7.2 Identification of Experts

Some approaches, such as Dominoes [33, 34], Emergent Expertise Locator [23], Expertise Browser [24], and Usage Expertise [31], aim to identify experts in software projects. Some of these approaches (Dominoes and Emergent Expertise Locator) are based on the approach by Cataldo et al. [9, 10], who developed a technique to measure task dependencies among people. They use matrices to represent various dependency relationships. From this, they aim to answer who must coordinate with whom to get the work done. Dominoes allows different kinds of explorations over matrices, and it can be used to identify experts for a given project or software artifact. Dominoes is capable of using GPU for processing operations, which enables the analysis of large-scale data. Emergent Expertise Locator produces a ranked list of the most likely emergent team members with whom to communicate, given a set of files currently deemed to be of interest. Expertise Browser identifies experts over regions of the code, such as modules or even subsystems by using the concept of: (1) Experience Atoms (EAs), which are basic units of experience in change management systems, and (2) the atomic change (delta) made to the source code or to the project's documentation. Finally, the concept of Usage Expertise is introduced to recommend experts for files, where the developer accumulates expertise not only by editing methods, but also by calling (using) them.

All these approaches extract information from the Version Control Systems and Issue Tracking Systems. Some of these systems are similar to TIPMerge, and are based on changes performed via commits; others check for different kinds of information, such as a method calls, opened and closed issues, etc. While these approaches all identify experts, they only take into consideration previous history, and do not discern changes in branches. As a result, equal weights are assigned to all files. However, in our situation we know that changes across branches and their dependencies might have a bigger impact on the merge decision than prior changes alone.

Other studies on identification of experts have focused on pull request assignment [18, 22, 36, 37]. Yu et al. [36, 37] proposed an approach that combines information retrieval with social network analysis to help project managers find a suitable reviewer for each pull request. Jiang et al. [18] proposed CoreDevRec to recommend core members for contribution evaluation in GitHub. CoreDevRec uses support vector machines to analyze different kinds of features, including file paths of modified codes, relationships between contributors and core members, and activeness of core members. De Lima Júnior et al. [22] proposed the use data mining to identify the most appropriate developers to analyze a pull request. They use a set of attributes and classification strategies to suggest developers to analyze pull requests.

These works are closely related to the recommendation of developers for branch merging, as they aim to recommend developers to verify the actual contribution and possibly merge it with the rest of the project. Nevertheless, in general, pull requests contain commits of a single developer and are small [14]. Moreover, the author of the pull request usually syncs their forked branch in advance to ease reintegration, making the process more like a workspace commit. In the more general case of merging branches, the number of developers, the syncing interval, and the number of commits per branch is variable and can be high in some situations.

8. CONCLUSIONS

This work, to the best of our knowledge, is the first to make developer recommendations for integrating branches. Our approach, implemented in TIPMerge, leverages historical information about changes in the branches as well as past history, and the dependency among files. We found that we perform the best in projects that either have no integrators (Category I) or have an integration team (Category II). We obtain the best accuracy at 62% for the top-1 recommendation (project Eureka) and a best accuracy at 98% for the top-3 recommendations (project Diploma). When we compare our results (top-3 recommendations) with the majority class, we get an improvement in predictions in most cases (24 out of 28 projects). Among the projects where we outperform the majority class, we have a normalized accuracy improvement of 30.7% (median) for the top-1 recommendation and a normalized accuracy improvement of 60.8% (median) for the top-3 recommendations. We further investigated the team contribution structures in the cases where TIPMerge had a decay (i.e., was worse than the majority class). Our exploratory analysis suggests that the role of developers (i.e., core team member, lead, QA, founder), as well as their skills (e.g., continuous integration) can affect who becomes responsible for the merge.

We performed interviews with three expert developers from two projects in our corpus. From our interviews, we found that factors like: (1) person performing the most recent change, (2) knowledge about specific parts of the code base, and (3) personal preference, had an effect on who was eventually responsible for the merge. In several cases where the top recommendation was incorrect, that developer had, in fact, participated in a collaborative merge or supported the merge developer in some fashion.

Our results suggest that TIPMerge can be further extended to incorporate the above factors into the analysis algorithm. We also plan to run the analysis at a finer grain (method level), as this will provide a detailed understanding of file dependencies and developer knowledge about specific parts of the code base. Further, we will extend the dependency calculation to also consider new dependencies added by changes in the branches. Finally, we intend to replicate this analysis over a larger corpus of projects.

In conclusion, our results suggest that TIPMerge can be useful in not only predicting the most appropriate developer to perform the merge when there is no integrator in the team, but also in recommending other developers who can support the integrator.

9. ACKNOWLEDGMENTS

This work is partially supported by CAPES (10614-14-1), CNPq, FAPERJ, and NSF CCF-1253786 and IIS-1314365. We also would like to thank STI/UFF for giving us access to their

projects during evaluation and José Ricardo da Silva Junior for helping us using Dominoes library.

10. REFERENCES

[1] Agrawal, R. and Srikant, R. 1994. Fast Algorithms for Mining Association Rules in Large Databases. *International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1994), 487–499.

[2] Anvik, J., Hiew, L. and Murphy, G.C. 2006. Who Should Fix This Bug? *International Conference on Software Engineering (ICSE)* (Shanghai, China, 2006), 361–370.

[3] Anvik, J. and Murphy, G.C. 2011. Reducing the Effort of Bug Report Triage: Recommenders for Development-oriented Decisions. *ACM Trans. Softw. Eng. Methodol.* 20, 3 (Aug. 2011), 10:1–10:35.

[4] Bang, J. young, Popescu, D., Edwards, G., Medvidovic, N., Kulkarni, N., Rama, G.M. and Padmanabhuni, S. 2010. CoDesign: a highly extensible collaborative software modeling framework. *The 32nd International Conference on Software Engineering (ICSE)* 2, 243-246.

[5] Berzins, V. 1994. Software Merge: Semantics of Combining Changes to Programs. *ACM Trans. Program. Lang. Syst.* 16, 6 (Nov. 1994), 1875–1903.

[6] C. Bird and T. Zimmermann, Assessing the Value of Branches with What-If Analysis. *ACM SIGSOFT Int'l Symp. Foundations of Software Eng. (FSE)*, 2012, pp. 45–54.

[7] Bird, C., Zimmermann, T. and Teterev, A. 2011. A theory of branches as goals and virtual teams. *4th International Workshop on Cooperative and Human Aspects of Software Engineering* (New York, NY, USA, 2011), 53–56.

[8] Brun, Y., Holmes, R., Ernst, M.D. and Notkin, D. 2011. Proactive detection of collaboration conflicts. *ACM SIGSOFT Int'l Symp. Foundations of Software Eng. (FSE)* (2011), 168–178.

[9] Cataldo, M., Herbsleb, J.D. and Carley, K.M. 2008. Socio-technical Congruence: A Framework for Assessing the Impact of Technical and Work Dependencies on Software Development Productivity. *The Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement* (New York, NY, USA, 2008), 2–11.

[10] Cataldo, M., Wagstrom, P.A., Herbsleb, J.D. and Carley, K.M. 2006. Identification of coordination requirements: implications for the Design of collaboration and awareness tools. *20th anniversary conference on Computer supported cooperative work (CSCW)*, 353–362.

[11] Costa, C., Figueiredo, J.J.C., Ghiotto, G. and Murta, L. 2014. Characterizing the Problem of Developers' Assignment for Merging Branches. *International Journal of Software Engineering and Knowledge Engineering*. 24, 10 (Dec. 2014), 1489–1508.

[12] Costa, C., Figueirêdo, J.J.C. and Murta, L. 2014. Collaborative Merge in Distributed Software Development: Who Should Participate? *The Twenty-Sixth International Conference on Software Engineering and Knowledge Engineering (SEKE)* (Vancouver, Canada, 2014), 268–273.

[13] Estler, H.C., Nordio, M., Furia, C.A. and Meyer, B. 2013. Unifying Configuration Management with Merge Conflict Detection and Awareness Systems. *2nd Australian Software Engineering Conference (ASWEC)* (Washington, DC, USA, 2013), 201–210.

[14] Gousios, G., Pinzger, M. and Deursen, A. van 2014. An Exploratory Study of the Pull-based Software Development Model. *International Conference on Software Engineering (ICSE)* (Hyderabad, India, 2014), 345–355.

[15] Guimarães, M.L. and Silva, A.R. 2012. Improving early detection of software merge conflicts. *34th International Conference on Software Engineering* (Piscataway, NJ, USA, 2012), 342–352.

[16] Hattori, L. and Lanza, M. 2010. Syde: a tool for collaborative software development. ACM/IEEE *32nd International Conference on Software Engineering* (May 2010), 235–238.

[17] Horwitz, S. 1990. Identifying the Semantic and Textual Differences Between Two Versions of a Program. *ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1990), 234–245.

[18] Jiang, J., He, J.-H. and Chen, X.-Y. 2015. CoreDevRec: Automatic Core Member Recommendation for Contribution Evaluation. *Journal of Computer Science and Technology*. 30, 5 (Sep. 2015), 998–1016.

[19] Kagdi, H., Gethers, M., Poshyvanyk, D. and Hammad, M. 2012. Assigning change requests to software developers. *Journal of Software: Evolution and Process*. 24, 1 (Jan. 2012), 3–33.

[20] Koegel, M., Naughton, H., Helming, J. and Herrmannsdoerfer, M. 2010. Collaborative model merging. *ACM International conference companion on Object oriented programming systems languages and applications companion (OOPSLA)* (New York, NY, USA, 2010), 27–34.

[21] Lautamäki, J., Nieminen, A., Koskinen, J., Aho, T., Mikkonen, T. and Englund, M. 2012. CoRED: browser-based Collaborative Real-time Editor for Java web applications. *ACM 2012 conference on Computer Supported Cooperative Work (CSCW)* (New York, NY, USA, 2012), 1307–1316.

[22] de Lima Júnior, M.L., Soares, D.M., Plastino, A. and Murta, L. 2015. Developers Assignment for Analyzing Pull Requests. A*CM Symposium on Applied Computing* (New York, NY, USA, 2015), 1567–1572.

[23] Minto, S. and Murphy, G.C. 2007. Recommending Emergent Teams. *Fourth International Workshop on Mining Software Repositories (MSR)* (Washington, DC, USA, 2007), 20-26.

[24] Mockus, A. and Herbsleb, J.D. 2002. Expertise Browser: A Quantitative Approach to Identifying Expertise. *24th International Conference on Software Engineering (ICSE)* (New York, NY, USA, 2002), 503–512.

[25] Murphy, G. and Cubranic, D. 2004. Automatic bug triage using text categorization. *The Sixteenth International Conference on Software Engineering & Knowledge Engineering* (2004), 209-2014.

[26] Nieminen, A. 2012. Real-time collaborative resolving of merge conflicts. *8th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)* (2012), 540–543.

[27] Oliva, G.A. and Gerosa, M.A. 2011. On the Interplay between Structural and Logical Dependencies in Open-Source

Software. *25th Brazilian Symposium on Software Engineering (SBES)* (Sep. 2011), 144–153.

[28] Pappa, G.L. and Freitas, A.A. 2006. Automatically evolving rule induction algorithms. *Machine Learning: ECML 2006*. Springer. 341–352.

[29] Portillo-Rodriguez, J., Vizcaino, A., Ebert, C. and Piattini, M. 2010. Tools to Support Global Software Development Processes: A Survey. *5th IEEE International Conference on Global Software Engineering (ICGSE)*. (2010), 13–22.

[30] Sarma, A., Redmiles, D. and van der Hoek, A. 2012. Palantir: Early Detection of Development Conflicts Arising from Parallel Code Changes. *IEEE Trans. Softw. Eng.* 38, 4 (Jul. 2012), 889–908.

[31] Schuler, D. and Zimmermann, T. 2008. Mining Usage Expertise from Version Archives. *International working conference on Mining software repositories (MSR)*. (New York, NY, USA, 2008), 121–124.

[32] Shihab, E., Bird, C. and Zimmermann, T. 2012. The Effect of Branching Strategies on Software Quality. *ACM-IEEE international symposium on Empirical software engineering and measurement (ESEM)*. (New York, NY, USA, 2012), 301–310.

[33] da Silva, J.R., Clua, E., Murta, L. and Sarma, A. 2015. Niche vs. breadth: Calculating expertise over time through a fine-grained analysis. *IEEE 22nd International Conference on Software Analysis, Evolution and Reengineering (SANER)* (Mar. 2015), 409–418.

[34] da Silva Junior, J.R., Clua, E., Murta, L. and Sarma, A. 2014. Exploratory Data Analysis of Software Repositories via GPU Processing. *The Twenty-Sixth International Conference on Software Engineering and Knowledge Engineering (SEKE)* (Vancouver, Canada, 2014), 495-500.

[35] Wloka, J., Ryder, B., Tip, F. and Ren, X. 2009. Safe-commit Analysis to Facilitate Team Software Development. *31st International Conference on Software Engineering (ICSE)*. (Washington, DC, USA, 2009), 507–517.

[36] Yu, Y., Wang, H., Yin, G. and Ling, C.X. 2014. Reviewer Recommender of Pull-Requests in GitHub. I*nformation and Software Technology*. (2014), 609–612.

[37] Yu, Y., Wang, H., Yin, G. and Ling, C.X. 2014. Who Should Review this Pull-Request: Reviewer Recommendation to Expedite Crowd Collaboration. *Software Engineering Conference (APSEC), 2014 21st Asia-Pacific* (Dec. 2014), 335–342.

[38] Zimmermann, T., Dallmeier, V., Halachev, K. and Zeller, A. 2005. eROSE: guiding programmers in eclipse. *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (2005), 186–187.

[39] Zimmermann, T., Weisgerber, P., Diehl, S. and Zeller, A. 2004. Mining Version Histories to Guide Software Changes. *The 26th International Conference on Software Engineering* (Washington, DC, USA, 2004), 563–572.